Karlsruher Institut für Technologie

Lehrstuhl für Programmierparadigmen
Sebastian Ullrich sebastian.ullrich@kit.edu
Max Wagner maximilian.wagner@kit.edu

Semantik von Programmiersprachen – SS 2019

http://pp.ipd.kit.edu/lehre/SS2019/semantik

Lösungen zu Blatt 7: Prozeduren Besprechung: 14.06.2019

1. Welche der folgenden Aussagen sind richtig, welche falsch? (H)

(a) $[(\texttt{p},\ \texttt{i := i * 1; call p})] \vdash \langle \texttt{call p},\ \sigma\rangle \Downarrow \sigma$

(b) $([(\texttt{p},\ \texttt{skip})],\ \texttt{call q})$ ist ein Programm der Sprache While_{PROC}.

(c) $\langle \texttt{skip},\ \sigma\rangle$ ist die einzige blockierte Konfiguration der Small-Step-Semantik für While_{PROC}.

(d) Wenn $P \vdash \langle c,\ \sigma\rangle \overset{\infty}{\to}_1$, dann enthält c eine `while`-Schleife.

(e) Sei $P \equiv [(\texttt{p},\ \texttt{y},\ \texttt{x := 4; result := 5 * y})]$, $E_0 \equiv [\mathsf{x} \mapsto 0, \mathsf{y} \mapsto 1]$ und $s \equiv [\mathsf{next} \mapsto 2]$.
Wenn $P, E_0 \vdash \langle \texttt{\{ var x = 3; y <- call p(x); y := y + x \}},\ E_0,\ s\rangle \Downarrow s'$,
dann $s'(E_0(\mathsf{x})) = 4$ und $s'(E_0(\mathsf{y})) = 18$.

(f) Wenn $P, E_0 \vdash \langle c,\ E,\ s\rangle \Downarrow s'$, dann $s'(\mathsf{next}) = s(\mathsf{next})$.

Lösung:

(1a) Falsch. `p` ist eine rekursive Prozedur, die *nie* abbricht. Wie bei unendlichen Schleifen gibt es für unendlich viele rekursive Aufrufe keine Ableitung.

(1b) Richtig. Die Syntax überprüft nicht, ob eine aufgerufene Methode existiert.

(1c) Falsch. Das Fortschrittslemma gilt *nicht* für While_{PROC} (zumindest nicht in der altbekannten Form). Beispielsweise ist auch die Konfiguration `call p` blockiert, falls es keine Prozedur `p` in der Prozedurdeklarationsliste P gibt, d.h., $P \vdash \langle \texttt{call p},\ \sigma\rangle \not\to_1$.
Auch die Big-Step-Semantik blockiert, wenn eine Prozedur nicht vorhanden ist. Dies bedeutet, dass Nichtableitung nicht mehr mit Nichttermination gleichgesetzt werden kann.

(1d) Falsch. Nichttermination ist auch durch Rekursion möglich.

$$[(\texttt{p}, \texttt{call p})] \vdash \langle \texttt{call p},\ \sigma\rangle \to_1 \langle \texttt{call p},\ \sigma\rangle \to_1 \dots$$

(1e) Richtig. Man zeichne einen Ableitungsbaum.

(1f) Richtig. Beweis mittels Regelinduktion über die Big-Step-Semantik.
- Fälle $\textsc{Skip}^{\text{P1}}_{\text{BS}}$, $\textsc{Ass}^{\text{P1}}_{\text{BS}}$, $\textsc{WhileFF}^{\text{P1}}_{\text{BS}}$: Trivial.
- Fälle $\textsc{Seq}^{\text{P1}}_{\text{BS}}$, $\textsc{IfTT}^{\text{P1}}_{\text{BS}}$, $\textsc{IfFF}^{\text{P1}}_{\text{BS}}$, $\textsc{WhileTT}^{\text{P1}}_{\text{BS}}$:
  Folgt direkt aus den Induktionsannahmen.
- Fälle $\textsc{Block}^{\text{P1}}_{\text{BS}}$, $\textsc{Call}^{\text{P1}}_{\text{BS}}$: Folgt direkt aus der Regel. □

2. Small-Step-Semantik für Prozeduren mit Parametern (H)

Für Prozeduren mit einem Parameter (While_{PROCP}) gibt es folgende Vorschläge für eine Aufrufregel der Small-Step-Semantik:

(a)
$$\frac{(p, x, c) \in P}{P \vdash \langle y\ \texttt{<- call}\ p(a),\ \sigma\rangle \to_1 \langle x\ \texttt{:=}\ a;\ c;\ y\ \texttt{:= result},\ \sigma\rangle}$$

(b) $$\frac{(p, x, c) \in P}{P \vdash \langle y \texttt{ <- call } p(a), \sigma\rangle \to_1 \langle \texttt{\{ var result = 0; \{ var } x \texttt{ = } a \texttt{; } c \texttt{ \}; } y \texttt{ := result \}}, \sigma\rangle}$$

(c) $$\frac{(p, x, c) \in P}{P \vdash \langle y \texttt{ <- call } p(a), \sigma\rangle \to_1 \langle \texttt{\{ var } x \texttt{ = } a \texttt{; \{ var result = 0; } c \texttt{; } y \texttt{ := result \} \}}, \sigma\rangle}$$

(d) $$\frac{(p, x, c) \in P \qquad z \text{ nicht verwendet in } P \text{ und } a}{P \vdash \langle y \texttt{ <- call } p(a), \sigma\rangle \to_1 \langle \texttt{\{ var } z \texttt{ = 0; \{ var } x \texttt{ = } a \texttt{; \{ var result = 0; } c \texttt{; } z \texttt{ := result \} \}; } y \texttt{ := } z \texttt{ \}}, \sigma\rangle}$$

Untersuchen Sie, in wie weit diese Regeln call-by-value Prozedurparameter und Rückgabewerte korrekt abbilden. Begründen Sie Ihre Antwort; ggf. mit Beispielprogrammen, die entsprechende Defizite aufzeigen. Welche dieser Regeln modellieren statische Variablenbindung?

Lösung: Alle vorgestellten Regeln haben Probleme.

(2a) Diese Regel ist nur die Erweiterung *dynamischer* Variablenbindung um einen Parameter.
Mangel: Parameter- und Rückgabewert überhaupt nicht lokal, kein call-by-value.
Beispiel: $P \equiv$ [(`p, x, x := 5; result := 1`)],
Aufruf: `{ var result = 0; x := 2; y <- call p(1); z := result }`
Nach dem Aufruf hat `x` den Wert 5, nicht 2; `z` den Wert 1, nicht 0.

(2b) Parameter- und Rückgabevariable werden hier außerhalb des Aufrufs wegen der Blöcke nicht geändert.

i. Die Reihenfolge der Blöcke ist schlecht gewählt. a kann `result` enthalten, das durch den umgebenden Block umgebunden wird.
Beispiel: $P \equiv$ [(`p, x, result := x`)],
Aufruf: `result := 1; y <- call p(result)`
Am Ende hat `y` den Wert 0, richtig wäre aber 1.
Lösung dafür: Man initialisiert nicht x auf a, sondern auf $\mathcal{N}^{-1}[\![\mathcal{A}[\![a]\!]\,\sigma]\!]$. Dann vermischt man aber mehrere Schritte in einem, was dem Small-Step-Gedanken entgegenläuft.

ii. Wenn y `result` ist, ist das y an den neuen Block gebunden, damit geht der Rückgabewert verloren.
Beispiel: $P \equiv$ [(`p, x, result := x`)],
Aufruf: `result := 1; result <- call p(5)`
Am Ende hat `result` noch immer den Wert 1, richtig wäre 5.

(2c) Im Gegensatz zu (2b) ist hier Blockreihenfolge besser, a wird zu Blockbeginn korrekt ausgewertet, Problem (2b) i. tritt nicht mehr auf. Dafür tritt das Problem (2b) ii. des verlorenen Rückgabewertes hier verstärkt auf, weil auch im (viel häufigeren) Fall $y = x$ der Rückgabewert verloren geht. Beispiel wie vorher.

(2d) Durch das Einführen einer neuen Zwischenspeichervariable z, die nicht in P oder a verwendet wird, verschwinden die Probleme mit versehentlichen Bindungen. Der Fall, dass x gleich `result` ist, bleibt weiterhin problematisch. In diesem Fall ist der Parameterwert nicht mehr zugreifbar. Das gleiche Problem hat übrigens auch die Big-Step-Semantik-Regel $\textsc{Call}^{\text{P1}}_{\text{BS}}$. Dort könnte man dies einfach lösen, indem das Speicherupdate für x nicht an der Stelle $s(\mathsf{next})$ durchgeführt wird, sondern nochmals über die neue Variablenbindung aufgelöst wird. Bei dieser Regel könnte man dies lösen, indem man die Blockreihenfolge z, `result` und x verwendet, z auf a und x auf z initialisiert – dann kann z auch in a vorkommen:

$$\frac{(p, x, c) \in P \qquad z \text{ nicht verwendet in } P}{P \vdash \langle y \texttt{ <- call } p(a), \sigma\rangle \to_1 \langle \texttt{\{ var } z \texttt{ = } a \texttt{; \{ var result = 0; \{ var } x \texttt{ = } z \texttt{; } c \texttt{; } z \texttt{ := result \} \}; } y \texttt{ := } z \texttt{ \}}, \sigma\rangle}$$

Wesentlicher Nachteil dieser Regel ist, dass sie die Einzelschritt-Semantik mehrdeutig macht – die Wahl von z ist nicht eindeutig. Die so entstehenden Programme sind aber semantisch alle äquivalent, nur der Determinismusbeweis wird wesentlich komplizierter. Da die Menge der Variablen in P endlich ist, kann auch leicht eine deterministische Auswahlfunktion angegeben werden.

Keine der Regeln eignet sich für statische Variablenbindung, da alle die Prozeduraufrufe textuell ersetzen und dabei keine Variablenumbenennung vornehmen (können). Da Funktionsaufrufe auch nur strukturierte `goto`s sind, können wir für echte statische Bindung die Continuation-Semantik aus dem letzten Übungsblatt wiederverwenden. Wir könnten sie mit der Speicherverwaltung aus der Bigstep-Semantik kombinieren, aber zur Abwechslung verwenden wir stattdessen die abstraktere Darstellung über einen expliziten Stack von Assoziativlisten, von denen jede die aktiven Variablendeklarationen eines aktiven Prozeduraufrufes speichert.

Wir benutzen also $\mathsf{Namespace} \equiv [\mathsf{Var} \times \mathbb{Z}]$ als Typ für Stackframes, zusammen mit zwei Funktionen $read : \mathsf{Namespace} \times \mathsf{Var} \rightharpoonup \mathbb{Z}$ und $write : \mathsf{Namespace} \times \mathsf{Var} \times \mathbb{Z} \to \mathsf{Namespace}$, welche eine lokale Variable im übergebenen Namespace lesen oder schreiben:

$$read([], x) = \bot$$

$$read(d \cdot ds, x) = \begin{cases} i & \text{falls } d = (x, i) \\ read(ds, x) & \text{falls } d = (y, i) \text{ mit } x \neq y \end{cases}$$

$$write(d \cdot ds, x, n) = \begin{cases} (x, n) \cdot ds & \text{falls } d = (x, i) \\ d \cdot write(ds, x, n) & \text{falls } d = (y, i) \text{ mit } x \neq y \end{cases}$$

Da *read* und *write* die Liste linear bis zum ersten passenden Eintrag durchsuchen, verschatten vordere Deklarationen andere weiter hinten. Verlassen wir einen Codeblock / eine Funktion, so muss der oberste Stackframe modifiziert oder verworfen werden[1]. Diese Operationen kodieren wir als `POP` und `y <- RET` in die Continuations hinein, somit brauchen wir also auch eigene Regeln, die diese Instruktionen verarbeiten. Der ursprüngliche Zustand σ wird nur noch für globale Variablen verwendet.

Ein Call wertet das Argument aus, und legt das Unterprogramm, einen neuen Namespace, und die dazugehörige `RET`-Operation auf die jeweiligen Stacks.

$$\frac{\begin{array}{c}\textsc{Call}_{\text{PSS}'}: \\ (p, x, c) \in P\end{array}}{P \vdash \langle y \texttt{ <- call } p\texttt{(}a\texttt{)} \cdot cs,\ s \cdot st,\ \sigma\rangle \to_1 \langle c \cdot \texttt{y <- RET} \cdot cs,\ [(x, \mathcal{A}[\![a]\!]\,\sigma[s]), (\texttt{result}, 0)] \cdot s \cdot st,\ \sigma\rangle}$$

Der Ausdruck $\sigma[s]$ steht dabei für den durch lokale Variablen partiell verschatteten globalen Zustand:

$$\sigma[s](x) = \begin{cases} read(s, x) & \text{falls } read(s, x) \neq \bot \\ \sigma(x) & \text{sonst} \end{cases}$$

Wird eine neue lokale Variable angelegt, muss eine entsprechende Deklaration in den Namespace der aktuellen Funktion gelegt werden, und deren spätere Löschung durch hinterlegen einer `POP`-Operation sichergestellt werden.

$$\textsc{Block}_{\text{PSS}'}: P \vdash \langle \texttt{\{ var } x \texttt{ = } a\texttt{; } c \texttt{ \}} \cdot cs,\ s \cdot st,\ \sigma\rangle \to_1 \langle c \cdot \texttt{POP} \cdot cs,\ ((x, \mathcal{A}[\![a]\!]\,\sigma[s]) \cdot s) \cdot st,\ \sigma\rangle$$

[1] Wer andere Veranstaltungen dieses Lehrstuhls besucht hat, erkennt dieses Schema geschachtelter Stacks vielleicht schon als *Symboltabelle*.

Die Assignment-Regel muss nun zwischen lokalen und globalen Zuweisungen unterscheiden. Das mag lästig erscheinen, bildet aber andererseits vielleicht passender ab, dass Zuweisungen auf lokale und globale Variablen im intuitiven Sinne nunmal "verschiedene Semantiken" haben.

$$\textsc{AssLocal}_{\text{PSS'}}\colon \frac{read(s,x) = n}{P \vdash \langle x \texttt{ := } a \cdot cs,\ s \cdot st,\ \sigma\rangle \to_1 \langle cs,\ write(s,x,\mathcal{A}\,[\![a]\!]\,\sigma[s]) \cdot st,\ \sigma\rangle}$$

$$\textsc{AssGlobal}_{\text{PSS'}}\colon \frac{read(s,x) = \bot}{P \vdash \langle x \texttt{ := } a \cdot cs,\ s \cdot st,\ \sigma\rangle \to_1 \langle cs,\ s \cdot st,\ \sigma[x \mapsto \mathcal{A}\,[\![a]\!]\,\sigma[s]]\rangle}$$

Schließlich sollten wir noch definieren, was `POP` und `RET` tun: `POP` löst eine Variablendeklaration in der aktuellen Funktion wieder auf, und `RET` entfernt den Namespace der soeben berechneten Prozedur. Da wir uns hier nicht darum kümmern wollen, ob das Resultat eines Prozeduraufrufes in eine lokale oder globale Variable gespeichert wird, delegieren wir der Einfachheit halber an die Zuweisungsregeln.

$$\textsc{Pop}_{\text{PSS'}}\colon P \vdash \langle \texttt{POP} \cdot cs,\ (d \cdot ds) \cdot st,\ \sigma\rangle \to_1 \langle cs,\ ds \cdot st,\ \sigma\rangle$$

$$\textsc{Ret}_{\text{PSS'}}\colon P \vdash \langle \texttt{y <- RET} \cdot cs,\ s \cdot st,\ \sigma\rangle \to_1 \langle \texttt{y := } \mathcal{N}^{-1}\,[\![\sigma[s](\texttt{result})]\!] \cdot cs,\ st,\ \sigma\rangle$$

Die übrigen Regeln der Small-Step-Semantik werden nicht groß von der Erweiterung auf Prozeduren gestört:

$$\textsc{Seq}_{\text{PSS'}}\colon P \vdash \langle c_0\texttt{;}\ c_1 \cdot cs,\ st,\ \sigma\rangle \to_1 \langle c_0 \cdot c_1 \cdot cs,\ st,\ \sigma\rangle$$

$$\textsc{IfTT}_{\text{PSS'}}\colon \frac{\mathcal{B}\,[\![b]\!]\,\sigma[s] = \mathbf{tt}}{P \vdash \langle \texttt{if (}b\texttt{) then } c_0 \texttt{ else } c_1 \cdot cs,\ s \cdot st,\ \sigma\rangle \to_1 \langle c_0 \cdot cs,\ s \cdot st,\ \sigma\rangle}$$

$$\textsc{IfFF}_{\text{PSS'}}\colon \frac{\mathcal{B}\,[\![b]\!]\,\sigma[s] = \mathbf{ff}}{P \vdash \langle \texttt{if (}b\texttt{) then } c_0 \texttt{ else } c_1 \cdot cs,\ s \cdot st,\ \sigma\rangle \to_1 \langle c_1 \cdot cs,\ s \cdot st,\ \sigma\rangle}$$

$$\textsc{While}_{\text{PSS'}}\colon$$
$$P \vdash \langle \texttt{while (}b\texttt{) do } c \cdot cs,\ st,\ \sigma\rangle \to_1 \langle \texttt{if (}b\texttt{) then } c\texttt{; while (}b\texttt{) do } c \texttt{ else skip} \cdot cs,\ st,\ \sigma\rangle$$

$$\textsc{Skip}_{\text{PSS'}}\colon P \vdash \langle \texttt{skip} \cdot cs,\ s \cdot st,\ \sigma\rangle \to_1 \langle cs,\ s \cdot st,\ \sigma\rangle$$

Übung: Was sollte der Startzustand dieser Semantik sein, was sind die blockierten Zustände?

3. Call by reference (Ü)

Die in der Vorlesung vorgestelle Big-Step-Semantik für Prozeduren mit einem Parameter wertet den übergebenen Parameter beim Aufruf aus und übergibt nur den Wert an die aufgerufene Prozedur. Insbesondere bleibt der Wert einer Variablen, die als Parameter übergeben wird, im aufrufenden Kontext unverändert. Diese Parameterübergabeart heißt *call by value*. Daneben gibt es auch noch *call by reference*, bei der Änderungen am Parameterwert in der aufgerufenen Prozedur auch nach dem Ende des Aufrufs in der aufrufenden Prozedur sichtbar sind. Passen Sie in dieser Aufgabe die Big-Step-Semantik von While_{PROCP} entsprechend an:

(a) Passen Sie die Syntax-Definition von While_{PROCP} an, so dass nur noch Variablen als Parameter verwendet werden können.

(b) Ändern Sie die Big-Step-Semantik so, dass Parameter immer als call by reference übergeben werden.

(c) Ändern Sie die Regeln erneut so, dass auch die Rückgabewertvariable mit call by reference übergeben wird.

(d) Finden Sie ein Programm, bei dem sich das Verhalten in beiden Varianten von call by reference und call by value unterscheidet. Belegen Sie dies durch die Ableitungsbäume.

Lösung:

(3a) Die Syntax muss die erlaubten Ausdrücke für Parameter in Prozeduraufrufe auf Variablen einschränken. Die Anweisungssyntax lautet:

$$\mathsf{Com} \quad c \quad ::= \quad \ldots \mid y \texttt{ <- call } p(x)$$

(3b) Es muss nur die Big-Step-Semantikregel angepasst werden:

$$\frac{(p,x,c) \in P \qquad P, E_0 \vdash \langle c,\ E_0[x \mapsto E(z), \mathtt{result} \mapsto s(\mathsf{next})],\ s[\mathsf{next} \mapsto s(\mathsf{next})+1]\rangle \Downarrow s'}{P, E_0 \vdash \langle y \texttt{ <- call } p(z),\ E,\ s\rangle \Downarrow s'[E(y) \mapsto s'(s(\mathsf{next})), \mathsf{next} \mapsto s(\mathsf{next})]}$$

(3c)

$$\frac{(p,x,c) \in P \qquad P, E_0 \vdash \langle c,\ E_0[x \mapsto E(z), \mathtt{result} \mapsto E(y)],\ s\rangle \Downarrow s'}{P, E_0 \vdash \langle y \texttt{ <- call } p(z),\ E,\ s\rangle \Downarrow s'}$$

Auch wenn diese Regel jetzt ohne next auskommt, braucht man dies trotzdem noch für die Blockregel $\textsc{Block}^{\mathrm{P1}}_{\mathrm{BS}}$.

(3d) Beispiel:

$$\begin{aligned} P &\equiv [(\mathtt{p},\ \mathtt{x},\ \mathtt{x := 1; result := 2; y := 3})] \\ V &\equiv [\mathtt{x},\ \mathtt{y}] \\ c &\equiv \mathtt{x := 4; y <- call p(x)} \end{aligned}$$

Bei call-by-value hat x am Ende den Wert 4, bei call-by-reference den Wert 1.

Die Ableitungsbäume:

$$\dfrac{\dfrac{}{P, E_0 \vdash \langle \mathtt{x := 4},\ E_0,\ s_0\rangle \Downarrow s_1} \qquad \dfrac{\dfrac{}{P, E_0 \vdash \langle \mathtt{x := 1},\ E_1,\ s_2\rangle \Downarrow s_3} \qquad \dfrac{\dfrac{}{P, E_0 \vdash \langle \mathtt{result := 2},\ E_1,\ s_3\rangle \Downarrow s_4} \qquad \dfrac{}{P, E_0 \vdash \langle \mathtt{y := 3},\ E_1,\ s_4\rangle \Downarrow s_5}}{P, E_0 \vdash \langle \mathtt{result := 2; y := 3},\ E_1,\ s_3\rangle \Downarrow s_5}}{P, E_0 \vdash \langle c_{\mathtt{p}},\ E_1,\ s_2\rangle \Downarrow s_5}}{P, E_0 \vdash \langle \mathtt{y <- call p(x)},\ E_0,\ s_1\rangle \Downarrow s_6}}{P, E_0 \vdash \langle c,\ E_0,\ s_0\rangle \Downarrow s_6}$$

call by value:

Umgebung:	**Belegung:**	x	y	result		
$E_0 = [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 1]$		0	1			
$E_1 = E_1[\mathtt{x} \mapsto s_1(\mathsf{next}), \mathtt{resultresult} s_1(\mathsf{next})+1]$		2	1		3	
Zustand:	**Werte:**	next	0	1	2	3
$s_0 = [E_0(\mathtt{x}) \mapsto ?, E_0(\mathtt{y}) \mapsto ?, \mathsf{next} \mapsto \lvert V \rvert]$		2	?	?		
$s_1 = s_0[E_0(\mathtt{x}) \mapsto 4]$		2	4	?		
$s_2 = s_1[s_0(\mathsf{next}) \mapsto \mathcal{A}[\![\mathtt{x}]\!](s_1 \circ E_0), \mathsf{next} \mapsto s_0(\mathsf{next})+2]$		4	4	?	4	?
$s_3 = s_2[E_1(\mathtt{x}) \mapsto 1]$		4	4	?	1	?
$s_4 = s_3[E_1(\mathtt{result}) \mapsto 2]$		4	4	?	1	2
$s_5 = s_4[E_1(\mathtt{y}) \mapsto 3]$		4	4	3	1	2
$s_6 = s_5[E_0(\mathtt{y}) \mapsto s_5(s_1(\mathsf{next})+1), \mathsf{next} \mapsto s_1(\mathsf{next})]$		2	4	2	*1*	*2*

call by reference für Parameter

Umgebung:	**Belegung:**	x	y	result	
$E_0 = [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 1]$		0	1		
$E_1 = E_1[\mathtt{x} \mapsto E_0(\mathtt{x}), \mathtt{result} \mapsto s_1(\mathsf{next})]$		0	1	2	

Zustand:	**Werte:**	next	0	1	2
$s_0 = [E_0(\mathtt{x}) \mapsto ?, E_0(\mathtt{y}) \mapsto ?, \mathsf{next} \mapsto \lvert V \rvert]$		2	?	?	
$s_1 = s_0[E_0(\mathtt{x}) \mapsto 4]$		2	4	?	
$s_2 = s_1[\mathsf{next} \mapsto s_0(\mathsf{next}) + 1]$		3	4	?	?
$s_3 = s_2[E_1(\mathtt{x}) \mapsto 1]$		3	1	?	?
$s_4 = s_3[E_1(\mathtt{result}) \mapsto 2]$		3	1	?	2
$s_5 = s_4[E_1(\mathtt{y}) \mapsto 3]$		3	1	3	2
$s_6 = s_5[E_0(\mathtt{y}) \mapsto s_5(s_1(\mathsf{next})), \mathsf{next} \mapsto s_1(\mathsf{next})]$		2	1	2	*2*

call by reference für Parameter und Rückgabe

Umgebung:	**Belegung:**	x	y	result
$E_0 = [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 1]$		0	1	
$E_1 = E_1[\mathtt{x} \mapsto E_0(\mathtt{x}), \mathtt{result} \mapsto E_0(\mathtt{y})]$		0	1	1

	Zustand:	**Werte:**	next	0	1
	$s_0 = [E_0(\mathtt{x}) \mapsto ?, E_0(\mathtt{y}) \mapsto ?, \mathsf{next} \mapsto \lvert V \rvert]$		2	?	?
$s_2 = s_1 =$	$s_0[E_0(\mathtt{x}) \mapsto 4]$		2	4	?
	$s_3 = s_2[E_1(\mathtt{x}) \mapsto 1]$		2	1	?
	$s_4 = s_3[E_1(\mathtt{result}) \mapsto 2]$		2	1	2
$s_6 = s_5 =$	$s_4[E_1(\mathtt{y}) \mapsto 3]$		2	1	3